C++

1108 서민준

함수 오버로딩 (1)

다른 매개변수를 가진 **같은 이름**인 여러 **함수**를 만들 수 있는 C++의 기능

```
int MyFunc(int num)
{
    return num + 1;
}

int MyFunc(int num1, int num2)
{
    return num1 + num2;
}
```

호출 형태를 보고
어떤 함수를 호출하는 건지
알 수 있지 않을까?

함수 오버로딩 (2)

C++ 컴파일러는 이름이 같은 함수가 정의되었을 때 다음 조건 중 하나라도 만족하면 오버로딩을 허용한다.

1. 매개변수의 자료형이 다르다!

```
int MyFunc(int num)
{
    return num + 1;
}

int MyFunc(double num)
{
    return num + 1;
}
```

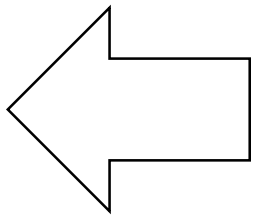
2. 매개변수의 수가 다르다!

```
int MyFunc(int num)
{
    return num + 1;
}

int MyFunc(int num1, int num2)
{
    return num1 + num2;
}
```

함수 오버로딩 (3)

함수의 반환형은 함수 오버로딩에 고려되지 않는다.

```
int MyFunc(int num)
{
    return num + 1;
}

double MyFunc(int num)
{
    return num + 1;
}
```

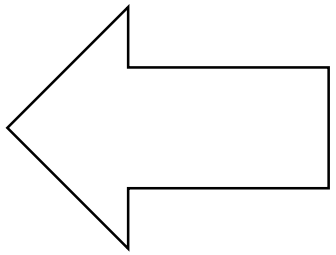


위에서 정의한 MyFunc의 재정의로 판단하고 **컴파일 오류**를 일으킨다.

매개변수의 디폴트 값 (1)

매개변수에 값을 전달하지 않았을 때 매개변수에 특정 값을 전달하고 싶을 때 사용하는 기능

```
int MyFunc(int num = 0)
{
    return num + 1;
}

int main()
{
    MyFunc();
    return 0;
}
```



이런 식으로 매개변수의 값을 전달하지 않았을 때는
자동으로 디폴트 값이 매개변수에 대입된다.

매개변수의 디폴트 값 (2)

디폴트 값은 **여러 개** 가질 수 있다.

```
int MyFunc(int num1 = 0, int num2 = 0)
{
    return num1 + num2;
}
```

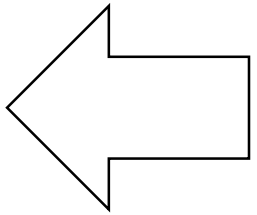
위 함수는 다음과 같은 형태로 호출될 수 있다.

```
MyFunc(); // num1, num2에 모두 디폴트 값 0이 전달됨.
MyFunc(1); // num1에는 1이, num2에는 디폴트 값 0이 전달됨.
MyFunc(2, 3); // num1에는 2가, num2에는 3이 전달됨.
```

매개변수의 디폴트 값 (3)

디폴트 값은 **오른쪽**에서부터 채워야 한다.

```
int MyFunc(int num1 = 0, int num2)
{
    return num1 + num2;
}
```

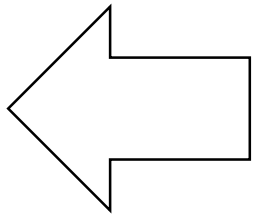


다음과 같은 정의는 컴파일 오류

선언된 후에는 디폴트 값을 다시 **선언할 수 없다**.

```
int MyFunc(int num = 0);
```

```
int MyFunc(int num = 0)
{
    return num + 1;
}
```

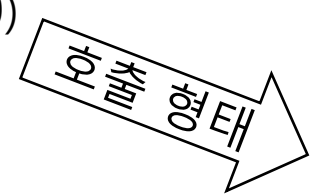


다음과 같은 정의는 컴파일 오류

매개변수 디폴트 값 (4)

함수 오버로딩이 가능하나 **주의**해야 한다.

```
int MyFunc(int num1 = 0, int num2 = 0)
{
    return num1 + num2;
}

int MyFunc(int num)
{
    return num + 1;
}

int MyFunc()
{
    return 0;
}
```




```
MyFunc(5, 6);
MyFunc(4);
MyFunc();
```

컴파일러가 어떠한 함수를 호출한 것인지 **구분할 수 없다**.

매크로 함수

#define SQUARE(x) ((x) * (x))

특징

전처리기가 처리한다.

장점

- 일반적인 함수의 비해 **실행속도가 빠르다**.

단점

- **정의하기 어렵다**.
- 정의에 **한계**가 있다.
- **디버그가 어렵다**.

inline 함수

```
inline int SQUARE(int x)
{
    return x * x;
}
```

장점

- 실행속도가 일반함수보다 빠르다.
- 정의하기 쉽다.
- 디버그가 쉽다.

단점

- 자료형에 의존적이다.

특징

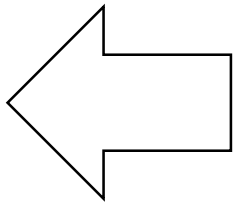
컴파일러가 처리한다.

함수가 인라인화에 해가 된다고 판단하면 inline 키워드를 무시

반대로 일반 함수도 인라인화했을 때 성능에 득이 된다고 판단하면 인라인화를 진행하기도 함

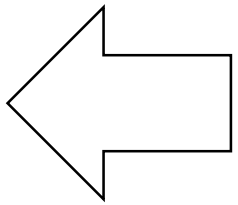
namespace (1)

```
int data;

int MyFunc(int num)
{
    return num + 1;
}
```



기본적으로 전역 namespace에 정의된다.

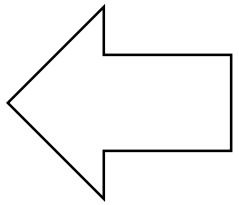
```
int MyFunc(int num)
{
    return num + 1;
}

int MyFunc(int num)
{
    return num + 1;
}
```



두 함수가 모두 전역 namespace에 정의되어 있다면
재정의 문제로 컴파일 오류가 발생한다.

namespace (2)

```
namespace Ex1
{
    int MyFunc(int num)
    {
        return num + 1;
    }
}

namespace Ex2
{
    int MyFunc(int num)
    {
        return num + 1;
    }
}
```



다음과 같이 정의하면 정의된 공간이 서로 다른 것으로 인식하여 아무런 문제를 발생시키지 않는다.

namespace (3)

```
namespace Ex1
{
    int data;

    int MyFunc(int num)
    {
        return num + 1;
    }
}
```

namespace 안에 정의된 변수나 함수를 사용하는 방법은 2가지가 있다.

1. 범위 지정 연산자(::)를 사용한다.

```
Ex1::data = 1;
Ex1::MyFunc(3);
```

2. using 명령문을 사용한다.

```
using namespace Ex1;

data = 2;
MyFunc(4);
```

namespace (4)

namespace 내에 있는 함수의 선언과 정의는 다음과 같이 분리한다.

```
namespace Ex1
{
    int MyFunc(int num);
}

int Ex1::MyFunc(int num)
{
    return num + 1;
}
```

namespace는 다른 namespace에 **중첩**될 수 있다.

```
namespace Ex1
{
    namespace Ex2
    {
        int MyFunc(int num)
        {
            return num + 1;
        }
    }
}
```

bool형

참과 **거짓**의 표현을 위한 키워드 **true**, **false**를 저장할 수 있는 **자료형**

```
bool isTrue = true;
bool isFalse = false;
```

true와 **false**는 각각 **1과 0을 의미하지 않는다**!

'참'과 '거짓 ' 을 나타내기 위한 1바이트의 크기의 데이터일 뿐입니다.
숫자와 연동시키지 말고 그냥 '참'과 '거짓 ' 을 나타내는 논리형으로 생각해주기 바란다.

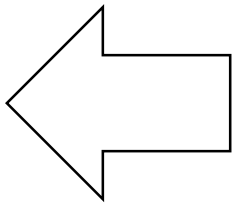
참조자 (1)

메모리 공간에 부여된 **이름 이외에 새로운 별명**을 붙여 **참조**하는 기능

사용 방법은 자료형**&** 별명 = 기존 변수명; 의 형식으로 사용된다.

```
int data = 1;
int& ref = data;
```

```
data = 2;
ref = 2;
```



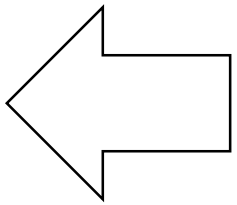
이 두 문장 모두 같은 결과를 보인다.

참조자 (2)

참조자 사용 방법에는 다음과 같은 특징이 있다.

1. 선언과 동시에 초기화해야 한다.

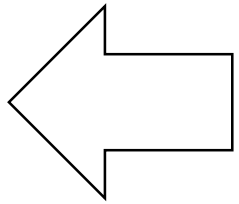
```
int& ref;
```



다음 문장은 컴파일 에러를 일으킨다.

2. NULL값으로 초기화할 수 없다.

```
int& ref = NULL;
```

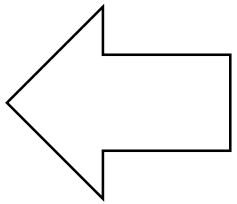


다음 문장은 컴파일 에러를 일으킨다.

참조자 (3)

3. 상수 값은 일반적으로 참조할 수 없다.

```
const int num = 0;
int& ref = num;

int& ref = 8;

int& ref = "C++";
```



다음 문장들은 모두 컴파일 오류를 일으킨다.

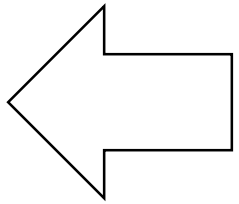
3-1. 단, const 참조자는 참조할 수 있다.

```
const int& ref = "C++";
```

참조자 (4)

4. 초기화 후에는 다른 변수를 참조하도록 변경할 수 없다.

```
int data1 = 1;
int& ref = data1;

int data2 = 4;
ref = data2;
```



다음 문장은 컴파일 에러를 일으킨다.

동적할당

C에서 malloc의 역할을 하는 C++의 예약어는 **new**이다.
C에서 free의 역할을 하는 C++의 예약어는 **delete**이다.

new로 할당하는 방법

```
int* ptr1 = new int; // int형 변수를 하나 힙 영역에 할당
int* ptr2 = new int[5]; // 크기가 5인 int형 배열을 힙 영역에 할당
```

동적할당한 메모리를 해제하는 방법

```
delete ptr1; // 힙 영역에 할당된 변수를 해제해달라고 메시지 전달
delete[] ptr2; // 힙 영역에 할당된 배열을 해제해달라고 메시지 전달
```

C++에서 C언어의 헤더파일 추가하기

#include <stdio.h> -> #include <cstdio>
#include <stdlib.h> -> #include <cstdlib>
#include <math.h> -> #include <cmath>
#include <string.h> -> #include <cstring>

앞에 c를 붙이고, 뒤에 .h를 지워주면 된다!

C++ 형식으로 헤더파일을 추가하면 함수 오버로딩의 기능을 이용해
자료형별로 정의되어 있는 함수들이 존재하여 자료의 손실을 막아준다.

int abs(int num) // 절댓값을 구해주는 함수

long abs(long num)
float abs(float num)
double abs(double num)
long double abs(long double num)

클래스

```
class Test
{
private:
    int num;
public:
    void MyFunc() { }
};
```

const 멤버 함수

```
class EasyClass
{
private:
    int num;
public:
    void InitNum(int n) { num = n; }
    int GetNum() const { return num; }
};
```

정보은닉

프로그래머도 사람입니다, 여러분. 실수는 누구나 할 수 있잖아요?

그래서 그런 **프로그래머의 실수를 막기 위해서 제한된 방법으로의 접근만 허용**해서 **잘못된 값 변경을 막고, 실수를 하였을 때 쉽게 발견**할 수 있도록 해야 합니다.

이걸 **정보은닉**이라고 합니다.

캡슐화

A라는 약은 재채기, 콧물, 기침 완화의 기능을 가지고 있다.

그런데 재채기, 콧물, 기침용 캡슐이 따로따로 존재한다면 얼마나 불편할까?

반대로 B라는 약은 A와 동일한 기능을 가지고 있는데,
하나의 캡슐을 먹으면 기능을 발휘할 수 있다.

B라는 약과 같이, 하나의 목적(B라는 약은 코감기 처방)에 대해서
둘 이상의 기능(재채기, 콧물, 기침)이 모여 하나의 목적을 달성하는 것.

이것을 우리는 **캡슐화**라고 부른다.

생성자

```
class Test
{
private:
    int num;
public:
    Test() { num = 0; }
    Test(int n) : num(n)
};
```

그런데 이런 형태의 객체 생성은 불가능합니다!

```
class Test { . . . };

int main()
{
    Test testObj(); // 생성 불가능!
}
```

멤버 이니셜라이저

```
class Test
{
private:
    int num;
public:
    Test(int n) : num(n) { }
};
```

디폴트 생성자

```
class AAA
{
private:
    int num;
public:
    int GetNum{ return num; }
};
```

이런 경우에도 객체 생성이 불가능합니다.

생성자의 불일치

```
class Test
{
private:
    int num;
public:
    Test(int n) : num(n) { }
};
```

```
int main()
{
    Test obj1(1108);
    Test obj2; // 불가능!
}
```

private 생성자

```
class AAA
{
private:
    int num;
    AAA(int n) : num(n) { }
public:
    AAA() : num(0) {}
    AAA& CreateInitObj(int n) const
    {
        AAA* ptr = new AAA(n);
        return *ptr;
    }
    void ShowNum() const
    {
        std::cout << num << std::endl;
    }
};
```

class AAA
{
private:
int num;
AAA(int n) : num(n) {}
};

소멸자

```
class Test
{
private:
    int* arr;
public:
    Test(int size) { arr = new int[size]; }
    ~Test() { delete[] arr; }
};
```

객체 배열 & 포인터 배열

Test arr[10]; // 객체 배열 생성

Test* parr[10]; // 객체 포인터 배열 생성

this 포인터

객체 자신을 가리키는 용도로 사용되는 포인터

다음과 같은 경우 멤버변수가 초기화되지 않는다.

```
class Test
{
private:
    int num;
public:
    Test(int num)
    {
        num = 105;
        this->num = 207;
    }
};
```

SelfRef 반환

SelfRef는 객체 자신을 참조할 수 있는 참조자를 의미한다.

```
class SelfRef
{
private:
    int num;
public:
    SelfRef(int n) : num(n) { }
    SelfRef& Adder(int n)
    {
        num += n;
        return *this;
    }
};
```

```
int main()
{
    SelfRef obj(3);
    ref.Adder(1).ShowTwoNumber().Adder(2).ShowTwoNumber();
    return 1;
}
```

복사 생성자

다음과 같은 상황에서 객체의 값을 복사할 때 호출되는 생성자

다음과 같이 클래스가 정의되어 있다고 가정해보자.

class Test { . . . };

그럼 다음과 같은 경우에 복사 생성자가 호출된다.

Case1 : 객체의 생성과정에서 다른 객체를 초기값으로 대입할 때

```
Test obj1;
Test obj2 = obj1;
```

복사 생성자

Case2 : 함수의 호출 과정에서 매개변수로 객체를 Call-by-Value 방식으로 전달할 때
Case3 : 함수의 반환형이 객체이며, 객체를 반환할 때

```
Test returnTestObj(Test obj) { return obj; }

int main()
{
    Test test;
    returnTestObj(test);
}
```

복사 생성자

```
class TwoNum
{
private:
    int num1; int num2;
public:
    TwoNum(int n1, int n2) : num1(n1), num2(n2) { }
    TwoNum(SoSimple& copy) : num1(copy.num1), num2(copy.num2) { }
};
```

그런데 이럴 때는 복사 생성자가 호출이 안 됩니다!

```
Test returnTestObj(Test obj) { return obj; }


int main()
{
    Test test;
    Test test = returnTestObj(test);
}
```

const 객체

다음과 같은 클래스가 정의되어 있다고 가정해보자.

class SoSimple { . . . . . };

다음과 같이 생성하면 해당 객체의 멤버변수 값 변경을 허용하지 않는다!

const SoSimple sim1;

이렇게 된 객체는 const 멤버함수 이외의 함수를 호출할 수 없다.
  -> 그래서 const 멤버함수로 만들 수 있으면 최대한 만들라는 것이다!

const와 함수 오버로딩

void SimpleFunc() { . . . . }
void SimpleFunc() const { . . . . }

다음 두 함수는 컴파일러가 다른 함수로 처리한다.
  -> 즉, const의 유무도 함수 오버로딩의 조건에 포함된다는 뜻이다!

friend

friend 선언을 하면 해당 클래스의 private 멤버를 외부에서도 접근할 수 있다.

```
class Boy
{
private:
    int height;
    friend class Girl; // Girl 클래스를 friend로 선언함

public:
    Boy(int len) : height(len) { }
};
```

위 예제의 경우 Girl 클래스 내에서는 클래스 Boy의 private의 멤버 height에 접근이 가능하다.
하지만, Boy 클래스 내에서 Girl 클래스 내에 private 멤버를 접근할려면
Girl 클래스 내에서 friend 선언을 해줘야 한다.

static

```
class SoComplex
{
private:
    static int cmxObjCnt;
};
int SoComplex::cmxObjCnt = 0; // static 변수 초기화
```

클래스 내에 정의된 static 멤버변수를 클래스 변수라고도 한다.

클래스 변수 및 클래스 함수는 객체별로 가지고 있는 것이 아니라
모든 객체가 공유하고 있다는 특징이 있다.

static이 붙은 멤버변수 혹은 멤버함수는 전역함수와 생성 및 소멸 시점이 동일하다.

static

static 멤버 변수 또는 함수는 다음과 같이 활용할 수도 있다.

```
class SoComplex
{
private:
    static int cmxObjCnt;
};
int SoComplex::cmxObjCnt = 0; // static 변수 초기화

int main()
{
    std::cout << SoComplex::cmxObjCnt << std::endl;
}
```

객체가 생성되지 않아도 따로 메모리 공간이 할당되기 때문에
해당 변수를 활용하고 싶다면 객체를 생성하지 말고 클래스의 이름으로 접근하는 것이 좋다.

const static

```
class SoComplex
{
private:
    const static int cmxObjCnt = 0;
};
```

다음과 같이 const static으로 선언된 클래스 변수의 경우 바로 초기화가 가능하다.

mutable

```
class TwoNum
{
private:
    int num1;
    mutable int num2;

public:
    void CopyToNum2() const
    {
        num2 = num1; // 컴파일 OK!
    }
};
```

const 함수 내에서도 값의 변경을 예외적으로 허용해주는 키워드이다.

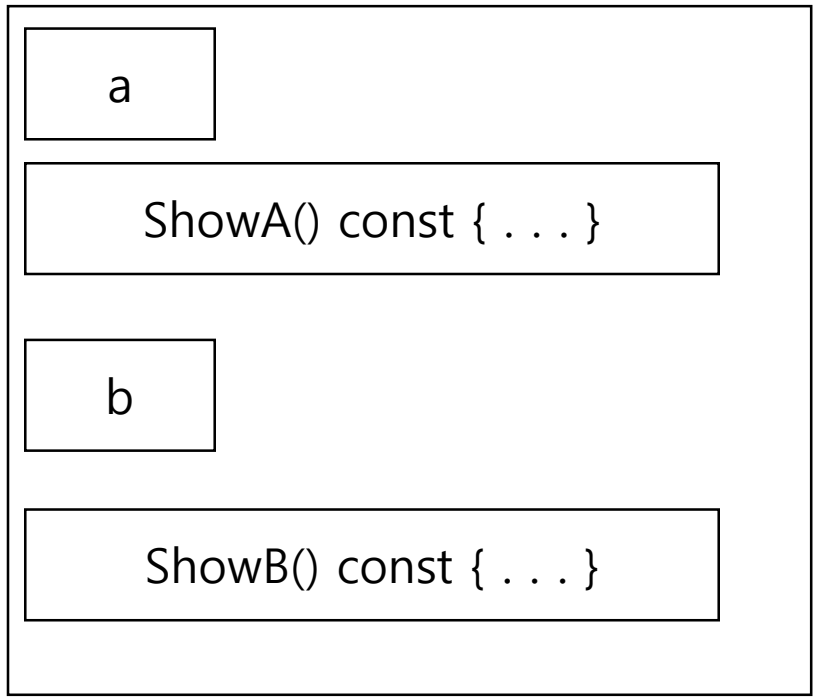
상속

기초(부모) 클래스의 모든 멤버를 물려받는 행위

```
class A
{
private:
    char a;
public:
    void ShowA() const { std::cout << a << std::endl; }
};

class B : public A
{
private:
    char b;
public:
    void ShowB() const { std::cout << b << std::endl; }
};
```

상속

유도(자식) 클래스 B의 메모리 구성



기초(부모) 클래스의 멤버

유도(자식) 클래스의 멤버

상속

다음과 같이 클래스가 정의되어 있다고 가정하자.

```
class Student { . . . };
class DSMStudnet : public Student { . . . };
```

이렇게 되었을 때, 유도(자식) 클래스는 기초(부모) 클래스의 생성자 호출의 의무가 있다.

```
UnivStudent(. . .) : Person(. . .)
{ . . . }
```

상속의 종류

```
class Base
{
private:
    int num1;
protected:
    int num2;
public:
    int num3;
}
```

유도 클래스

멤버 / 상속 방식	num1	num2	num3
private	접근 불가	O	O
protected	접근 불가	O	O
public	접근 불가	O	O

외부

멤버 / 상속 방식	num1	num2	num3
private	접근 불가	접근 불가	접근 불가
protected	접근 불가	접근 불가	접근 불가
Public	접근 불가	접근 불가	O

IS-A 관계

상속을 할 때는 반드시 다음의 관계를 만족해야 한다.

IS-A 관계

예)

부모 클래스 사람, 유도 클래스 고등학생이 정의되어 있을 때,
class 고등학생 : public 사람 { . . . };

이 상속은 **“사람은 고등학생이다.”**라는 문장이 성립되므로 좋은 상속이라 할 수 있다.

객체 포인터의 참조 관계

```
class Person { . . . };
class Student : public Person { . . . };
class HighSchoolStudent : public Student { . . . };
```

다음 관계에서 최상위 클래스 (Person 클래스)는 다음 문장이 모두 성립한다.

```
Person* ptr1 = new Person();
Person* ptr2 = new Student();
Person* ptr3 = new HighSchoolStudent();
```

C++에서 AAA형 객체 포인터는 AAA형 객체 또는
AAA를 직접 또는 간접적으로 상속하는 모든 객체를 가리킬 수 있다.

객체 포인터의 멤버 접근 관계

```
Class Person
{
Private:
    bool life;
};

Class Student : public Person
{
Private:
    std::string schoolName;
};

Class HighSchoolStudent : public Student
{
Private:
    bool dropout;
};
```

AAA형 객체 포인터는 AAA 또는 AAA를 직접 또는 간접적으로 상속하는 모든 객체를 가리킬 수 있다.

단, 접근 가능한 멤버는 클래스 AAA에 정의되어 있는 멤버만 가능하다.

함수 오버라이딩

```
class A
{
private:
    int num1;
public:
    void Show() const { std::cout << num1 << std::endl; }
};

class B : public A
{
private:
    int num2;
Public:
    void Show() const { std::cout << num2 << std::endl; }
};
```

가상함수

AAA형 객체 포인터는 AAA 클래스에 정의된 멤버에만 접근이 가능하다.
-> 그런데 오버라이딩한 함수는 내가 각 클래스별로 다르게 작동하도록 만든 거 아닌가?

그래서 C++은 virtual(가상)함수를 지원한다.

이렇게 하면 포인터의 자료형이 아니라
포인터가 가리키고 있는 객체의 자료형에 따라 멤버에 접근이 가능하다.

가상함수

```
class A
{
private:
    int num1;
public:
    virtual void Show() const { std::cout << num1 << std::endl; }
};

class B : public A
{
private:
    int num2;
Public:
    void Show() const { std::cout << num2 << std::endl; }
};
```